(19) Europäisches Patentamt
Eur pean Patent Office
Office européen des brevets

(11) EP 0 727 739 A1

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
21.08.1996 Bulletin 1996/34

(51) Int. Cl.6: G06F 9/44

(21) Application number: 95102234.2

(22) Date of filing: 17.02.1995

(84) Designated Contracting States:
DE FR GB

(71) Applicant: INTERNATIONAL BUSINESS MACHINES CORPORATION
Armonk, N.Y. 10504 (US)

(72) Inventors:
- Hollberg, Ulf, Dr.
D-69168 Wiesloch (DE)
- Ehrlich, Ralf Stefan
D-76131 Karlsruhe (DE)
- Scheere, Ulrich
D-71069 Sindelfingen (DE)
- Zimmermann, Olaf
D-69207 Sandhausen (DE)

(74) Representative: Schäfer, Wolfgang, Dipl.-Ing.
IBM Deutschland
Informationssysteme GmbH
Patentwesen und Urheberrecht
70548 Stuttgart (DE)

(54) Object-oriented programming interface for developing and running network management applications on a network communication infrastructure

(57) Disclosed is a programming interface for converting network management application programs written in an object-oriented language into network communication protocols. The application programs manipulate managed objects specified in e.g. GDMO/ASN.1 ISO standards.

Further disclosed are methods for mapping from GDMO templates and ASN.1 defined types into C++ programming language.

The interface particulary comprises object interface composing means for generating code which provides proxy managed object classes as local representatives for managed object classes and run time system means for providing proxy agent object classes as representatives for remote agents.

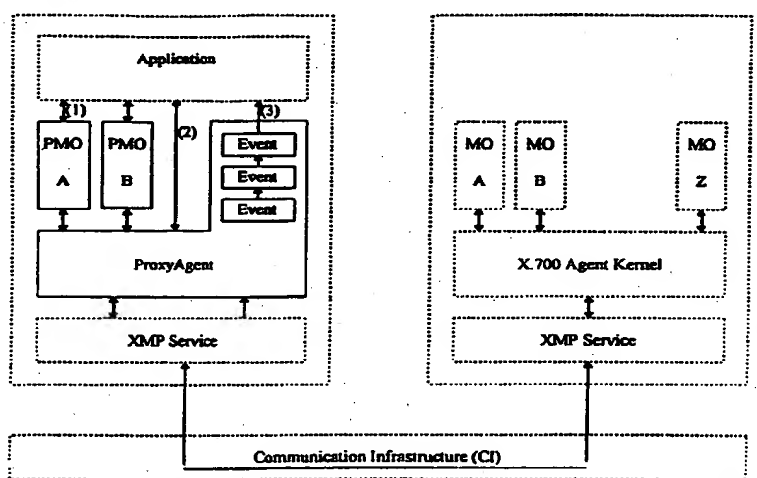

Fig. 1

EP 0 727 739 A1

Descripti n

This invention is directed to a programming interface for developing and running network management application programs written in an object-oriented language having object class definitions, on a network communication infrastructure wherein the application programs manipulate managed objects being specified in e.g. the GDMO/ASN.1 ISO standards and are made available at remote management agents through the communication infrastructure. Beyond this it relates to methods for mapping from GDMO templates and ASN.1 defined types into the C++ language and a platform for the implementation of the interface.

OSI network management applications and CCITT Telecommunication Management Network (TMN) applications are based on the ability to manipulate Managed Objects which are specified in GDMO/ASN.1 and which are made available at remote management agents through a communication infrastructure.

Currently, XMP/XOM from the X/Open [X/Open XMP] is the only standardized API to the communication infrastructure for management applications. XMP/XOM is cumbersome to use. XMP/XOM based applications are lengthy and difficult to write, understand and debug. Furthermore XMP/XOM does not allow for static (compile time) type checking, so that many type errors show up at run-time. Therefore most programmers certify that using XMP/XOM is cumbersome and time consuming. Implementers of network management applications are thus confronted with the user unfriendliness of the XMP/XOM interface.

In order to promote code quality and reusability more and more applications are written in the object-oriented programming language C++. Even though management information is defined in the object-oriented specification language GDMO, XMP/XOM uses the C language.

Further, managed objects are formally specified in GDMO and ASN.1. Development tools that support GDMO and ASN.1 can thus drastically reduce the development time of network management applications. Therefore a demand for a C++ embedding to hide the intricaci s of XMP/XOM and GDMO based tools to support the development of OSI management applications is ascertainable.

Th development of applications within the OSI management framework [ISO 10040] is a rather complex undertaking. The estimated costs for the development of new applications support this perception. In order to boost the development process, additional support by higher-level interface and corresponding tools is required.

It is therefore an objective of the invention to develop an object-oriented interface (OOI) which provides a genuine, object-oriented abstraction of OSI management information and services for use in regular, non-distributed applications.

A further objective of the invention is to provide an OOI for the access to managed objects which is as simple to use as possible.

Further objectives of the invention are to relieve the application programmers from most technical details related to communication and XMP/XOM, to provide an object-oriented, strong typed language embedding of manegement information and management services into C++, to generate automatically methods to manipulate specified managed objects, and to be open to future management paradigms or communication infrastructures, such as OSF-DME.

The requirements for the OOI design thus can be summarized as follows:

1. Relieve the application programmers from most technical details related to communication and XMP/XOM;

2. provide an object-oriented, strong typed language embedding of management information and management services into C++;

3. automatically generate methods to manipulate specified MOs; and

4. be open to future management paradigms or communication infrastructures, such as OSF-DME.

These problems are solved by the features of the invention laid down in the independent claims. The programming interface (OOI) according to the invention provides access to managed objects via telecommunication networks. The Object Interface Composer (OIC) automatically generates C++ class definition and implementation files based on Managed Object specifications written in GDMO and ASN.1 and thus increases the efficiency of program developers. Using the OOI, a network management application can access Managed Objects stored at remote agents through methods of those generated classes.

The intricacies of XMP/XOM are hidden from the application programmer by C++ classes. As a result application programmers can concentrate on writing their application instead of having to deal with communication protocols or low level interfaces to the communication stack. They can thus work in the application domain where their skills lay. The OOI shall hide the intricacies of the communication infrastructure and particulary of XMP/XOM behind a programmer-friendly object-oriented C++ operator interface.

As opposed to XMP/XOM based code, OOI based code is concise and readable. The OIC may also comprise means for minimizing the number of generated classes and the number of objects to be handled by an application at run time i.e.. generates C++ classes for the relevant GDMO templates only. The OOI therefore drastically simplifies the development of management applications by hiding the XMP API below C++ objects.

Furthermore the full embedding of Managed

Objects into C++ allows for strong type checking at compile time, whereas cumbersome debugging is usually necessary for XMP/XOM based applications. Without the OOI, programmers either use the cryptic and C-oriented XMP/XOM API or develop some kind of OOI on their own. Such "ad-hoc" solutions take time to develop and usually lack the support of a source code generator similar to the OIC so that the managed object specification must be manually translated. Those solutions are of course time consuming and error prone. With the OOI, the additional development effort and the weaknesses of ad-hoc solutions can be avoided. The OOI Run Time System provides C++ classes which allow convenient access to the Common Management Information Service (CMIS).

Both, the object-oriented interface (OOI) for the use in OSI management applications and the related Object Interface Composer (OIC), minimize the effort needed to build the communication related functions of management applications.

An application written on top of the OOI is independ nt of the management service provider. The current version of the OOI is based on the XMP/XOM [XMP] service, but future versions of the OOI could use a different communication vehicle such as OSF-DME. The application could be ported to a new service provider with a minimal effort. The OOI API depends not upon XMP/XOM so that applications must not be rewritten when the OOI is ported to another communication infrastructure.

Preferred embodiments of the interface according to the invention are characterized in the subclaims. The OOI shall provide static type checking and be easy to use. The OSI Definition of Management Information is object-oriented, thus the OOI takes advantage of object-oriented design techniques and provide a genuine object-oriented interface written in C++.

Managed Objects (MO) are formally described in the GDMO/ASN.1 language. This allows for the automatic generation of MO specific source code. The Object Interface Composer (OIC) takes GDMO/ASN.1 conformant MO specifications and generates C++ classes that provide methods to manipulate these objects. The OOI further provides methods to manipulat standardized MOs.

Strong typing is commonly defined as the compile time checking of type compatibility in programs; it is frequently used as co-notation of 'static typing'. This means that a correctly compilable program in strong typed language, such as C++, will be guaranteed to be type safe. Type safeness means, that variables have a defined type which completely specifies the value range and the permissible operations on the values of the type. Also, constants must be defined as specific values of certain types. This argument also applies to the type checking of parameters of procedures.

The net effect of strong typing is that the compiler will detect and prohibit the invokation of undefined methods on variables and illegal assignments of values of type X to variables of type Y. Illegal means here that no appropriate type cast has been defined explicitly.

For use with object-oriented languages strong typing is of even higher importance because in these languages it is common to define many application oriented types. In written distributed applications, debugging is far more complicated than for local programs. Without strong typing, errors may be caused by unintended misuse of defined variables. Obviously, the avoidance of these errors saves debugging time.

Further, type safeness is essential for applications which shall be installed in a wide range of network conditions. Using strong typing, the compiler is enabled to perform the compatibility checks for assignment, procedure parameters etc. If the compiler does not guarantee type safe programs, the type safeness must be enforced at run-time by checking the type compatibility at "the right spots" in the program, which is by itself an error prone task. The execution time for these run-time checks may reach a non-trivial percentage and thus degrade the performance of the application.

The OOI supports strong, static typing for management applications which work with a known inventory of management information. In addition, the generic part of the OOI supports generic management applications. Finally, to allow the coexistence of generic and strong typed component within the same application, the OOI makes provision for using the same objects through the type safe and the generic interface. This means that by using the OOI, objects will be allocated and used in a strong typed manner as long as their types are known at compile time. In addition, objects of types which are unknown at compile time, may be allocated and used via the 'weak' typed interfaces.

The invention is also related to methods for mapping GDMO templates and ASN.1 types into C++ classes. These subject-matters of the invention and the programming interface itself will become clear with regard to preferred embodiments of the invention illustrated by the appended drawing, wherein

Figure 1 shows the OOI components and their runtime environment;

Figure 2 gives a tabular overview of GDMO templates and their intended use;

Figure 3 shows a flow-chart of the GDMO/ASN.1 compilation process; and

Figure 4 is an example of a inheritance structure for DMI managed object classes.

The OOI design is based on the following object-oriented abstractions of the major constituents of OSI management:

1. Management information is represented by Managed Objects,

Notifications, and
ASN.1 types.

2. Management services is provided by Proxy Agents.

These abstractions allow the OOI to provide an easy to use programming interface. Furthermore, they separate the OOI implementation from the application, thus allowing for several different OOI implementations which are based on different communication infrastructures, to be exchanged transparently to the application.

Figure 1 shows the run-time environment of the OOI according to the invention. The OOI components are drawn with solid lines. An application can interact with the OOI through Proxy Managed Objects (PMO) (arrow 1), directly through the Proxy Agent objects (arrow 2), or through the notification event queue (arrow 3). The OOI uses the XMP API to access the Communication Infrastructure (CI) which allows it to communicate with an agent that implements the Managed Objects (MOs).

The ProxyAgent provides the Common Management Information Service (CMIS) as standardized by the ISO [ISO 9595 (CMIS)]. ProxyAgents are C++ classes which hide the C-oriented XMP API.

Proxy Managed Objects (PMO) are local representatives of remote managed objects. ProxyMOs are instances of C++ classes that are automatically generated by the Object Interface Composer (OIC). PMOs provide methods for strong typed access to the ASN.1 values of the attributes of managed objects and to the parameters of actions.

Incoming notifications are stored in an event queue. Notifications are instances of C++ classes that are automatically generated by the Object Interface Composer (OIC). Notification classes provide methods for the strong typed access to the ASN.1 values of the information and reply syntax of notifications.

The ASN.1 values of GDMO attributes, of GDMO action information parameters and of notification information and reply syntax are represented by instances of ASN.1 Type C++ classes which are also automatically generated from the ASN.1 definitions parsed by the OIC. The ASN.1 Type C++ classes provide a set of methods to manipulate the values of the ASN.1 type.

The OOI Run Time System (RTS) and the Object Interface Composer (OIC) thus offer utmost development support for those applications. The OOI Run Time System (OOI RTS) provides easy to use C++ classes to access Management Information and Management Services (XMP/XOM). The OIC and the RTS are closely related; in fact the code generated by the OIC must be linked to the OOI RTS to become executable.

The use of strong-typed local representations of remote managed objects and the generation of Proxy Managed Object (PMO) classes with the Object Interface Composer causes a paradigm shift from weakly-typed message-oriented communications programming to strongly-typed local object-oriented programming. This will increase the productivity of regular programmers and enable more programmers to develop management applications.

The OOI provides the following features:

1. Supports management applications written in C++

2. Uses GDMO and ASN.1 definitions as abstract object definitions

3. Uses automatically generated C++ classes from GDMO/ASN.1 definitions (done by the OIC)

4. Relieves the application developer from intricacies of communication interfaces

5. Separates the application from communication interfaces and technologies

6. Provide strong and weak type interface support

7. Provide run-time type information Meta Information

8. Offer a generic communication class with CMIS functionality, called the ProxyAgent

9. Leaves open the migration path towards the future communication architectures such as CORBA from OMG.

These features will be detailed in the following sections.

Mapping of GDMO templates into C++ Classes

GDMO defines several templates for the definitions of management information. Documents such as DMI or M3100 define managed objects with those templates. The Object Interface Composer (OIC) parses GDMO Managed Object definition documents (such as DMI) and generates C++ classes that represent the managed objects. This section briefly describes the templates defined in [ISO 10165-4(GDMO)] and explains how Managed Objects defined with those templates are mapped into C++ classes by the OIC.

Figure 2 gives an overview of the GDMO templates and their intended use. The OIC provides great flexibility for the generation of C++ classes for objects defined using the GDMO templates, so that important design decisions had to be taken. We decided not to generate one class for every usage of any template in the parsed document because of the huge number of classes that would have been generated using this approach. Instead, the OIC was configured to minimize the number of generated classes and the number of objects to be handled by the application at run time.

The OIC generates C++ Classes for the relevant GDMO templates only. Managed Objects are the most relevant objects for management application. A C++ class is generated for every GDMO Managed Object Class. The C++ classes reflect the inheritance hierarchy defined in the GDMO document.

The major interest of application writers is to get or set the values of the attributes of managed object instances, and to perform actions on them. Generating classes for GDMO Packages and GDMO Attributes would force the application to traverse two additional objects to get access to the value of an attribute.

But no classes are generated for GDMO Packages and GDMO Attributes. Instead, each managed object class provides methods to manipulate its attributes. Attributes have values which can be complex structures defined in ASN.1. A C++ class is generated for each attribute type defined in the GDMO/ASN.1 document parsed by the OIC. These classes provide methods to manipulate the attribute values.

Access methods to attributes are generated as methods of the managed objects classes of the managed objects that contain the attribute. A C++ class is generated for each ASN.1 type.

These classes provide methods to manipulate the values of the attributes.

Also no classes are generated for actions. Instead access methods for actions are generated as methods of managed objects without further indirection.

Notifications may arrive more or less unexpectedly at the management application and contain structured information of some types defined in ASN.1. A reply information structure may have to be transferred as a possible confirmation to the notification. Therefore, a C++ class is generated for every GDMO Notification template. This class provides appropriate access methods to the structured information. Confirmable notifications have a reply()-method. The optional attribute identifiers are used to generate additional access methods. The errorReply() method allows to return appropriate error information to the notifications issuer.

Parameters are not represented by classes. Parameters are rarely used and parameter information can alternatively be transferred through ASN.1 syntax.

Name bindings are not represented by classes. We regard name binding information to be of low relevance for management applications.

The Abstract Syntax Notation One (ASN.1) is used by GDMO to define all values which are transmitted between management applications and agents. As mentioned above, C++ classes are generated for all ASN.1 types.

The following restrictions are introduced to the design to improve the usability and the performance of the OOI: The value clauses limits the value range of GDMO attributes of managed objects. These clauses are of importance to agent implementers but not to application implementers. The OOI could have performed run time checking on the attribute values ("within range?"), but this checking has to be done in the agent so that we estimated that the performance cost was not justified. The value clause is therefore ignored by the OIC.

GDMO packages are regarded as aid for the definition of managed object classes. According to the GDMO standard, they are of no interest to management applications at run-time, because the attributes, actions and notifications which are defined within packages must be treated as properties of the managed object classes themselves [ISO standard 10165-4 (GDMO)].

GDMO attribute templates point (at least indirectly through another attribute) to the type of their value defined in ASN.1, assign an Object Identifier to this type and list the operations to be made available for the applications. The type information is kept in the GDMO Attribute Meta objects. The value is made accessible directly by the managed object, thus avoiding a superfluous hop and a separate run time object.

Strong and Weak Typed Usage

In order to support generic applications that can handle any object as well as specific applications that are tailored to handle a well known subset of the objects, all objects can be accessed in strong typed and in weak typed fashion.

The weak typed interface can be used to manipulate objects whose type is not (yet) known, e.g. analyzing the result of a scoped get (a scoped management request returns the management information for several managed object instances as a list of generic MO instances).

The strong typed interface should be used whenever possible to allow the compilers to detect type errors that would otherwise result in CMIP-ERRORS (or core dumps in an application that directly uses XMP/XOM) and to avoid time consuming run-time type checking that affects performance.

Both types of interfaces can be used interchangeably and concurrently within the same application.

Proxy Agents

The ProxyAgent is one of the fundamental abstraction of the OOI. The proxy agent provides the Common Management Interface Service (CMIS). A ProxyAgent acts as a proxy for a real, remote agent. ProxyAgents objects are local to the management applications. Agents are not aware of the existence of ProxyAgent objects. ProxyAgent objects hide the XMP-session and the XMP-context C- structure and the related XMP operations behind convenient methods of the ProxyAgent class.

Proxy Managed Objects

Proxy Managed Objects (PMO) are (stateless) representations of Managed Objects that are instantiated in

agents. Each PMO C++ class provides a set of object class specific methods through which a management application can conveniently submit CMIS requests to query or manipulate the real Managed Object in the agent. A management application typically instantiates an instance of a PMO class for each real Managed Object that it wishes to interact with.

Meta Information

Meta-information is a notion for type information which is made available at run time. For the OOI, the presence of meta information is essential to support the mixed usage of strong and weak typed interfaces. The meta information will most likely be used for the conversion between the binary and string representation of objects. In addition, the meta information of ASN.1 obj cts is used for encoding and decoding of their values.

F r the OOI, every GDMO/ASN.1 object has a pointer to its meta information object. All instances of on class share the same instance of meta information object.

The OOI-Environment

The OOI-Environment object has a single instance in the applications, in order to cluster those objects which belong to the OOI, e.g. proxyAgents or meta information objects. The OOI-Environment becomes visible to the programmer at initialization time and when the application should wait for the first event which happens on any of the existing proxyAgent objects, i.e. on any of the active XMP sessions.

GDMO/ASN.1 Object Interface Composer

The Object Interface Composer (OIC) is a tool for the generation of source code based on the specifications of management information in GDMO and in ASN.1. It takes its input from managed object class definiti ns written in accordance to the ISO standard "Guidelines for the Definition of Managed Objects" (GDMO) [ISO 10165-4 (GDMO)] and generates C++ classes (header and implementation files) for the managed objects, ASN.1 types and notifications defined in the selected document. The Object Interface Composer (OIC) therefore serves as a GDMO/ASN.1 Compiler generating C++ classes for XMP/XOM from GDMO/ASN.1 definitions.

The OIC is based on the IBM TMN-WorkBench/6000 [WorkBench]. GDMO and ASN.1 documents are parsed and stored in an relational database or in a shared library by the Managed Object Compiler (MOC) of the WorkBench. The Workbench then provides the GDMO and ASN.1 information through an API.

The OIC generates:

a C++ class for every GDMO Managed Object Class;

a C++ class for each ASN.1 type;

a C++ class for every GDMO notification;

meta information data structures for GDMO and ASN.1;

a set of utility files.

The GDMO/ASN.1 Compilation Process is shown in greater detail in Figure 3 and is described in the following.

ProxyAgent objects

ProxyAgent objects are local to the management applications. Agents are not aware of the existence of ProxyAgent objects. ProxyAgent objects hide the XMP-session and context C-structure and the related XMP operations behind convenience methods of the ProxyAgent class.

The ProxyAgent implementation provides synchronous and asynchronous methods. Synchronous methods do not return control to the application until a request is fully processed. Using synchronous OOI methods, a single process application blocks for an undetermined time while a CMIP request is being processed. This behavior may be appropriate for very simple applications, but not for an application that is user-interactive.

Asynchronous methods return control to the application as soon as a CMIP request is sent.

ProxyAgent objects provide a service interface to the create, get, set, action, cancelGet and delete operations of CMIS.

This service is used internally by the OOI implementation of proxy managed objects and by generic management applications which want direct access to a CMIS interface without using the proxy managed object abstraction. This interface is not directly used by applications which access managed object information through the Proxy Managed Object (PMO) abstraction.

Incoming notifications are queued in the event queue of the responsible proxy agent. The application can thus process notification at its leisure. The OOI optionally can trigger an application callback upon receipt of a notification.

Two distinct implementations of ProxyAgent for direct addressing (DA) and for non direct addressing (NDAPA) are provided. A Direct Addressing ProxyAgents (DAPA) can connect to one specific agent at a time. DAPAs can be used by management applications which communication with one specific agent. DAPAs are implemented in the ProxyAgentDA C++ class. Non Direct Addressing ProxyAgents (NDAPA), are not connected to specific agents. For each management request the agent must be addressed in one of two ways: Explicitly by supplying an addressing parameter as part of the request (the XMP responder-title) by means of a context object, and implicitly through the ORS. In this case the XMP (and the postmaster) determines the agent's address with the help of the ORS

directory service based on the object-class and the object-instance information of the request. NDAPAs are implemented by the ProxyAgentNDA C++ class.

Direct Addressing Proxy Agents (DAPA)

A DAPA object represents a real, remote agent. The connect() method with appropriate parameters will establish a connection (XMP-Session) to this agent. The disconnect-methods will release that session binding.

The creation of a Direct Addressing ProxyAgent object in a management application for use with a specific agent does neither imply, that this agent exists, nor that it can be connected. No verification is done when the DAPA object is created. The initial state is disconnected.

The management application must explicitly try to connect the DAPA object to a real agent. This attempt may fail. If the connect succeeded, the DAPA object is in stat connected and is able to transmit management requests to that agent. Internally, the connection between the DAPA and the agent is based on a XMP-Session.

The management application can explicitly disconnect a DAPA from the agent. A connection can also be aborted by the agent or the by the Management Information Service provider. The DAPA object is then in the "disconnected" state and can be reconnected to the same agent or to any other agent.

Non Direct Addressing ProxyAgents (NDAPA)

A management application can instantiate only one indirect addressing ProxyAgent object. The successful creation of an NDAPA does not imply, that there is an agent available for communication. No verification is done when the NDAPA is created. The management application must explicitly try to connect to the postmaster demon process. Using the connect() method, a non-direct addressing XMP-Session is established. This session remains active until the management application or the postmaster closes it. As long as the NDAPA object is in connected state, it can be used to communicate with any agent.

Mixing DAPA and NDAPA

Several DAPA objects and one NDAPA object may exist in the same management application at the same time. Each of the connected proxy agents has a connection (XMP session) with the agent. The OOI design intends that management application should only connect one ProxyAgent object to a specific real agent. If the application tries to use two DAPA objects to communicate with the same agent, or uses a DAPA and the NDAPA to communicate with the same agent, the noticeable effects are strictly dependent on the behaviour of XMP and the postmaster. In such cases it is possible that EFD's created over one proxy agent object cause notifications to appear on a different XMP-session and consequently, in the event queue of a different proxy agent object.

Operations Provided by ProxyAgents

Management operation can be performed on one or more attributes of one or more objects. The proxyAgents provide the full set of CMIS services with all parameters as defined in the standard [ISO 9595 (CMIS)]. The resulting structure of the argument and result parameters of the CMIS operations of the proxy agent interface are complex. Therefore, a set of additional methods "simple-create", "simple-get", "simple-set", "simple-action" and "simple-delete" is provided with less and simplified parameter to perform operations on only one attribute of one managed object or on several attributes of a single managed object.

The following methods to inquire the state and properties of a proxy agent are provided:

The connect() method establishes an XMP binding between the proxy agent and an agent or the postmaster.

The disconnect() method terminates the XMP binding between the proxy agent and an agent or the postmaster.

The isConnected() method checks whether the proxy agent is in the connected state or not.

The id() method returns the local id for the proxy agent. The id can be used to distinguish this instance from other instances of proxy agents within the same application.

The method fileDescriptor() returns the file descriptor (e.g. in AIX) which is associated with the XMP session. The AIX file descriptor is needed for advanced applications which want to write their own AIX 'select' call, e.g. to synchronize between OOI and Xwindows.

The reset() method terminates all activities associated with the proxy agent and re-establishes its initial state, including a disconnect().

The following methods to access and modify the underlying XMP data structures are provided:

XMPSession() returns a reference to the XMP session which is associated with the proxy agent instance.

contextControls() returns a reference to the XMP context object.

sessionControls() returns a reference to the XMP session object.

setPresentationModule() will replace the defined presentation module.

The following methods to inquire the state and properties of a proxy agent are provided:

dump() prints out the complete internal status of the Proxy Agent instance.

dumpRequestQueue() prints out the elements of the queue that stores requests (to be) sent to the agent.

dumpCompletedQueue() prints out the element of the queue of requests to which the agent has replied.

The following methods offer the full CMIS functionality:
MCreate() creates a managed object instance at an agent's site
MDelete() deletes one or more managed object instance at an agent's site
MGet() gets attribute values of one or several managed objects from an agent
MSet() replaces the values of attributes of one of several managed object at an agent
MAction() invokes an action of one or several managed objects at an agent.

The following methods are provided for easy to use synchronous CMIS functionality:
simpleMCreate() creates a managed object instance at an agent's site
simpleMDelete() deletes a managed object instance at an agent's site
simpleMGet() gets one attribute of a managed object from an agent
simpleMGetSome() gets some attributes of a managed object from an agent
simpleMSet() replaces one attribute of a managed object at an agent
simpleMSetSome() replaces some attribute of a managed object at an agent
simpleMAction() invokes the action of a managed object at an agent.

The following methods to wait for the completion of a request or a notification are provided:
wait() waits for a specified amount of time;
poll() checks with XMP whether something has arrived.

The following methods to inspect to local state of the proxy agent are provided:
HasNotificationQueue() checks whether this instance has a notification queue, i.e. is prepared to receive notifications
notificationQueue() returns a reference to the notification queue of the proxy agent
requestQueue() returns a reference to the request queue of the proxy agent
completedQueue() returns a reference to the completed queue of the proxy agent.

The Event Queue

The ProxyAgent objects contain an externally visible EventQueue-object where the received notifications are stored as typed objects (see arrow 4 in Figure 3). Notifications are received at any time when the proxy agent is receiving messages from its XMP session. The management application may process the notifications in the queue at any time.

The event queue is optional. A different constructor can be used for proxy agents whose role does not include monitoring so that they will never receive any notifications. The notifications are represented by typed-notification objects. They are inserted into the event queue of their proxyAgent instance as soon as they arrive at the XMP-session. Notification objects remain in the event queue, until they are explicitly deleted by the delete() method. For confirmable notifications, the application should invoke the errorReply() or the reply(). method. before the delete() method, otherwise the agent waiting for the confirmation might get confused.

For direct addressing ProxyAgents, the source of the notification is the specific agent, whereas for indirect addressing ProxyAgent, the source can be any agent (excluding those for which an direct addressing proxyAgent with role monitoring exists in the same application). The requestor-address of the sending agent and the requestor-title of the sending agent are not available from XMP.

For asynchronous requests, a request-object is allocated by the application and passed to the OOI. This object contains a list, which will be used to collect all replies to this particular request, regardless whether the replies are successful results or error results. The application explicitly passes control to the OOI RTS by invoking a method to check upon or to wait for the reception of incoming messages.

Wait Methods

Since several ProxyAgent objects may exist at the same time in the same application, several wait methods are offered:

The global wait method returns if anything was received on any Proxy agent object, (i.e. on any XMP-session).

The wait method of the proxy agent returns if anything was received on that session.

The wait method of the request object returns if anything was received on that request object.

It has to be distinguished between a single-event-mode wherein only a single incoming response or notification indication is received, added to the related queue and returned to the application, and a wait-for-completion-mode wherein partial replies to outstanding request will not cause the end of the wait method. A completed request or a notification will end the wait of the application.

Request Objects

Request objects represent requests which the application intends to send or has sent to a remote agent. These objects contain all the information needed to keep track of the request (as far as possible), to synchronize with the reply and to access the results or error information.

Request objects must be explicitly created and deleted by the application. Request objects can be reused several times.

The following methods to inquire the status of a request object are offered:

confirmationMode() will return a reference to the

actual confirmation mode of the request,
toBeConfirmed() checks whether the confirmation mode is set 'confirm'
wait Mode() distinguishes between the single event of completion mode for wait methods
invokeld() returns the invoke id
state() returns the processing state, e.g. 'outstanding'
state As String()returns the processing state as a string
isIdle() predicate on the state, is it 'idle' ?
isOutstanding() .. 'outstanding' ?
isCompleted() ... 'complete' ?
completionState() returns the completion state,
completionStateAsString() .. as string
isNormallyCompleted() predicate on the completion stat ,
isCouldNotBeIssued() .. due to a local error
isAbandonnedByUser() the user has aborted the request
isAbortedByProvider() the service provider, e.g. XMP aborted the request
errorOccurred() checks if an error did occur
numberOfResultElementsReceived() returns the number of results in the result queue
numberOfServiceErrorsReceived() returns the number of service errors encountered during processing
numberOfNonServiceErrorsOccurred() returns the number of other errors
The following methods to update data members (if the request is not in state 'outstanding') are provided
setWaitMode() will set the value of the wait mode of the request object
reset() will abort any outstanding activity and re-establishes the initial state of the object
abandon() aborts the outstanding activity by calling the XMP abandon method, including
cancelGet() in case of a get request. depending on its wait mode
wait() waits for a partial result or for the completion of the request depending on its wait mode
hasAttribute() checks whether the request did return an attribute with the passed OID
receiveAttribute() receive an attribute with the passed OID
receiveNextAttribute() is an iterator method
receiveActionReply() receives the reply of an action (if the request was to execute an action)
dump() formats the actual state of the request object into an 'ostream' object.

Request objects must be explicitly deleted by the application, even if the related proxy agent is deleted. All response queue elements included in the request object are automatically deleted with the request object. Additi nal incoming responses are also deleted.

An application may delete a response queue after all results have been received by means of the class destructor. An abandon/cancel operation on the outstanding operation does not delete the response queue (the queue has to be deleted explicitly).

Responses can not be received anymore after the ProxyAgent object was deleted or disconnected from the communication system either by the application or by failure.

Callbacks for the Reception of Incoming Messages

When using several asynchronous request at the same time, replies may appear in any order. To facilitate the processing of arriving reply messages, the OOI offers the possibility to define callback methods, which will be activated as soon as a reply message or a notification has been received. The OOI is single threaded, therefore callbacks are invoked only during wait() or poll() calls and not while the application is processing.

The OOI distinguishes four different tasks for reply callbacks, and therefore the possibility to register four different callbacks per request:
partialReply() will be called for every successful linked reply message from XMP
errorReply() will be called for every unsuccessful reply message from XMP
completed() will be call upon reception of the 'final' reply from XMP
disconnected() will be called during the disconnect processing which might have been triggered by XMP or the application.

Incoming responses for pending requests are represented by objects which have been derived from the ASN.1 definitions of CMIP. These object are put into the reply queue of the request object. Incoming notifications are represented by objects which also have been derived from the ASN.1 definitions of CMIP. These object are put into the event queue of their proxy agent. Then the callback method incomingNotificationCallback() which is defined by the application for the notification queue is executed with the notification object as parameter.

In general, callbacks have 'post-receival' semantics. The callback informs the application, that something has been received, and that the queue structures were updated. Thus when the callback is invoked, the object is already in the queue.

Partial response for an outstanding request

1. The response object is added to the response list of the request object. This includes updating of all related information of the request object, 'numberOfResponse' information.

2. Then, the partialReplyCallback() or errorReplyCallback() is invoked.

Final response for an outstanding request

The following steps are taken in addition to the partial response callback:

1. The request object is updated. Its state is changed to 'completed'. There is no 'final response object' added to the response queue.

2. The request object is moved from the requestQueue to the completedQueue

3. The final callback requestCompletedCallback() is called.

Incoming notification

1. The notification object is added to the notification queue. This includes updating the information in the notification queue header.

2. The incomingNotificationCallback() callback is invoked. This scheme allows the application to modify the queue structures, e.g. to 'unlink' the received data from the response objects in order to avoid copying. Some higher level receive methods may modify the queue structure, e.g. methods like getSubordinates, which convert the data of the response queue into a list of proxy MO's.

Mixed Processing of Synchronous and Asynchronous Requests

It is assumed that the application has issued one or more asynchronous CMIS requests. It then decides to send out a synchronous request. During the synchronous request, the complete application waits. In the meantime, responses for the asynchronous requests or notifications may arrive.

In order to receive the reply for the synchronous request, the OOI must receive any message from the XMP-Session of the proxy agent object which was used for the synchronous call. All callback methods defined for the incoming messages will be executed to preserve the semantics and to guarantee the highest responsiveness possible.

It can be argued whether the same approach should be used for the other proxy agents with outstanding requests. For those, we have decided against the receival to avoid unnecessary pre-reception of messages.

Flow control

The OOI design tries to avoid re-implementing functionality that is already covered by lower layers in the communication stack. Therefore, the OOI relies on the flow control mechanism of XMP and of any other underlying components. The application is responsible for being responsive enough for retrieving the data fast enough from XMP. Otherwise purging on XMP level and below may occur. There is a recommendation related to the IBM XMP implementation, which recommends to receive as much data as available as fast as possible. The OOI does not do internal buffering to avoid uncontrollable memory consumption in the manager application.

Using the asynchronous OOI, the manager application has all the mechanisms needed to receive messages from the agents as quickly as it seems advisable from the viewpoint of the application. The OOI receives messages from XMP during the processing of one of the several wait methods. Depending on the properties of the outstanding request objects and the kind of messages which arrive, one or more messages are received from one or more XMP-Session. The provision of callback functions which can handle every message from XMP as soon as it arrives, gives maximal control to the application.

Proxy Managed Objects

Specific management applications will be designed with and rely upon the specific knowledge of managed object classes and of their attributes, which have been defined in GDMO and ASN.1 prior to the development of the application. For those applications, we intend to offer utmost development support: The managed object and attribute templates defined in GDMO are automatically compiled into concrete classes with complete implementation in C++.

Proxy Managed Objects (PMO) are stateless representations of Managed Objects that are instantiated in agents. PMO C++ classes provide a set of methods through which a management application can conveniently call CMIS requests to query or manipulate the real Managed Object in the agent. A management application typically instantiates an instance of a PMO class for each real Managed Object that it wishes to interact with. PMOs may also be created as a result of OOI methods, e.g. getSubordinates().

The Object Interface Composer (OIC) generates a Proxy Managed Object (PMO) C++ class for every Managed Object Class (MOC) defined in the processed document. Each generated PMO class provides type-safe methods for the access to the mandatory and optional attributes and for the execution of the actions of the managed object. Type safe methods enforce strong typing and make up the "strong typed" interface of the generated PMO.

In addition to the strong typed methods, every PMO inherits from the OOIProxyMO class a set of "generic" methods which we call the "weak typed" interface of the PMOs. These methods are intended for management applications or components which do not know at compile time, which classes of managed objects might appear from some agent at run time. It should be noted that these generic methods can also be used for generated PMOs, but will execute less efficiently due to the necessary dynamic type checking done at run time.

The inheritance relation between generated PMO classes in C++ reflects the inheritance relation of the MOCs defined in GDMO/ASN.1 documents. In Fig. 3 an

example of an inheritance structure for DMI MOCs is shown. As can be seen in Fig. 3, all generated, strong typed PMO Classes are derived from the DMI_ProxyTop PMO class which in turn is derived from the OOIProxyMO class. All methods of the OOIProxyMO and DMI_ProxyTop classes are thus inherited by all proxy managed object classes.

The OOIProxyMO and OOIGenericMO classes were hand coded as part of the OOI run-time environment. The DMI_ProxyTop class and every of its subclasses are generated by the OIC.

The OOIGenericMO class is used to handle managed objects whose types are not known at compile time. This feature allows to write generic applications or to provide for the handling of new objects that will be defined after the application was completed.

Storage Management

F r the storage management of the OOI, it is assumed that the OOI will allocate objects and pass them to the application. The application must release all allocated objects which it received from the OOI. The OOI will only managed those objects which are used only internally and which are not made visible to the application. Upon reception of data from XMP, only the OOI knows its type, and must therefore allocate the object of the correct type, and after passing of such objects to the application, the OOI does not know when the application has finished using them.

Hereby the destructor of the OOI objects will take care of the proper deletion of contained objects. All methods are inherited from the OOIProxyMO class according to the semantic of the C++ language.

Constructors and Destructors of OOIProxyMO

Two constructors are available for OOIProxyMO objects; both expect as parameter the agent on which the MO resides and a pointer to the metainfo object. One of these additionally accepts the distinguished name of the object as parameter. In any case, the name can be set explicitly by the setMOInstance() method. Both Direct Addressing Proxy Agents (DAPAs) and Non Direct Addressing Proxy Agents (NDAPAs) can be used as parameter. The copy constructor and the assignment constructor have been explicitly disabled for the OOIProxyMO class and its subclasses. Copy constructors are not provided to avoid the automatic generation of multiple instances of proxy managed objects of the same managed object instance within one application.

Destructors are provided for each ProxyMO subclass. Both, constructors and destructors report errors through the exception mechanism.

The following methods are inherited from the OOIProxyMO class by each PMO class:
the setAgent() method can be used to overwrite the reference to the ProxyAgent object.
the setMOInstance() method can be used to overwrite the distinguished name.
the << print operator creates a formatted printout of the MOC and Managed Object Instances (MOI) values of the PMO.
the agent() method returns a pointer to the ProxyAgentXX object on which the MO resides,
the moClass() method returns a reference to the CMIS_ObjectClass (the MOC may not be known if the object is the result of a scoped MGet operation),
the moClassId() method returns the name of the class as a reference to an OOIString,
the moInstance() method returns the distinguished name of the MO as a reference to a CMIS_ObjectInstance,
the metaInfo() access method retrieves the run time Meta information (i.e. structural information specified in the GDMO MO class definition, see MIG.)
the hasConditionalPackage() access methods can be used to determine the presence of a conditional package (see below),
the reset() method re-establishes the initial status of the object.

Strong Typed Methods of PMO

The OIC generates C++ header and implementation files for every managed object class defined in the parsed GDMO document. Each PMO class provides type safe methods for the access to attributes and for the execution of the actions of the managed objects. Those methods are said to be strong typed.

As for their superclass, the strong typed PMO classes have disabled default, copy and assignment constructors. The constructor for a typed PMO expects the proxy agent on which the MO resides and the distinguished name of the MO instance as parameter; both are defaulted to NULL and can be modified later on by using the local utility methods setAgent and setMOInstance, which have been described above. The class of the represented MOC is known by the type of the proxyMO. Constructors and destructors report errors through the exception mechanism.

Multiple Inheritance

GDMO allows the definition of managed object classes being derived from more then one superior class (multiple inheritance). This section describes how the OOI represents multiple inheritance of managed object classes in the generated C++ classes.

The basic properties of the OOI representation are:

1. The class hierarchy of the OOI proxy managed object classes strictly follows the inheritance structure imposed by the GDMO definition. This includes multiple inheritance.

2. The same holds for the representation of meta information: in case of multiple inheritance, a meta

info object for a managed object class has multiple 'superior' references.

Even though C++ offers multiple inheritance, C++ has some serious restrictions. A simple mapping of GDMO multiple inheritance to C++ multiple inheritance is not feasible as will be explained in the sequel.

As long as multiple inheritance is used in order to inherit from different base classes only, C++ works fine. However, in case of a common base class for different inheritance paths, C++ problems arise. In order to have only one instance of the common base member variables, which is what is normally needed, the base class has to be made a 'virtual' base class. Then however, C++ no longer supports casting between base class pointers and sub-class pointers.

This restriction is not acceptable for the OOI, since for internal reasons (decoding), as well as for the user model (which is to support generic and type safe usage in a mixed fashion), the ability to cast pointers is a 'must'. Furthermore, experiments have shown, that today's C++ compilers impose a very large size overhead per instance.

As described above, the main problem of C++ is not related to multiple inheritance itself, but to the use of 'virtual' base class annotation. Thus the basic approach is to avoid this annotation, and to handle the 'virtual base class' property by other means. The original purpose of making the base class virtual is to avoid having multiple instances of the base class members.

In case of OOI proxy managed objects, this property is needed. Duplicated instances of the 'agent-reference' or 'packages-cache' members within a proxy managed object are unacceptable. The OOI approach is to move the 'data members' of a proxy managed object ut of the proxy managed object class into a separate object called 'proxy managed object data' (PMOData). This PMOData object is purely local and completely owned by its corresponding proxy managed object. The original proxy managed object merely contains a pointer to this PMOData object. As required for casting, the original proxy managed object class is NOT declared as a virtual base class. Obviously, in case of multiple inheritance this may lead to having duplicated pointers to the PMOData. The OOI runtime system guarantees, that all these pointers do point to the very sam object during the lifetime of the PMOData object.

The implementation uses the 'use-count' paradigm for the PMOData objects: during usage, the 'use-count' is equivalent to the number of pointer members of the related proxy managed object, and thus to the number of inheritance paths of a specific managed object class to a common base class.

To the user, this solution is completely hidden. The user is not aware of the existence of multiple pointers, nor of the fact that the proxy managed object data is stored in a separate object. All data and all operations are directly accessible from the proxy managed object interface.

Casting for the C++ proxy managed object classes is achieved by the OOI solution described above. However, in case of multiple inheritance, C++ casting requires to specify the exact casting path (at least at those places with multiple inheritance paths.). To simplify this, the OOI offers (as for other classes) a narrow()-operator, which allows to cast towards subclasses.

In addition, the OOI provides for the proxy managed object classes a widen()-operator for casting towards the 'OOIProxyMO' base class. The narrow() operator optionally performs run-time checking, whereas this is not necessary for the widen() operator. Thus there is no need to use plain C++ casts directly.

<u>Notifications</u>

Specific management applications rely upon the specific knowledge of notification object classes which have been defined in GDMO and ASN.1 prior to the development of the application. For those applications, we intend to offer utmost development support:

The notification templates defined in GDMO are automatically compiled into concrete classes with complete implementation in C++.

It is now assumed that Notification Objects are generated from GDMO/ASN.1 definitions by the Object Interface Composer (OIC) and are sent to a manager by means of a CMIS event report. The OOI RTS receives notifications and stores them in the EventQueue of the responsible proxy agent object.

<u>ASN.1</u>

Specific management applications will be designed with and rely upon the specific knowledge of GDMO/ASN.1 definitions, which have been defined in ASN.1 prior to the development of the application. For those applications, we intend to offer utmost development support: The abstractly defined ASN.1 types are automatically compiled into concrete classes with complete implementation in C++.

The specification language ASN.1 (for "Abstract Syntax Notation 1") has been defined by the ISO to specify the format of transmitted data in a formal, abstract notation [ASN1]. A standardized encoding scheme, such as the "Basic Encoding Rules" (BER) specifies the precise sequence of "bits on the wire". Thus two communicating partners are able to understand each other if they exchange data defined that is defined in ASN.1 and encoded according to the same encoding rules.

ASN.1 offers primitive types, string types and constructors which can be used to define further, application related types. Table 2 shows the types available in ASN.1. In addition to those types, ASN.1 offers the possibility to define named values for some types, and to define several kinds of subtypes.

Mapping Principles

As shown in Table 1 there is a set of primitive ASN.1 types and a set of constructors, which are used to compose application oriented complex types.

Base Library: For every primitive ASN.1 type and every ASN.1 constructor, there is a corresponding class in the ASN.1 C++ library, e.g. ASN1_Integer, ASN1_SetOf.

Application Classes: Each application defined ASN.1 type is mapped to one or several C++-classes. In the general case, instances of these classes will form a tree structure with instances of ASN.1 constructors as intermediate nodes and instances of primitive ASN.1 typ s as leaves. The root object of such a tree will be the application defined class which inherits from the outermost ASN.1 constructor class or simple type.

Common Methods: The generated C++ classes and those in the library are derived from a single common class 'ASN1_Type'. The declaration of functions such as assignment, comparison, print, encoding, decoding, checking, conversion into and from ASN.1 value notation, etc. as virtual methods in the common base class allows for generic usage of all ASN.1 specific C++ classes.

Strong & Weak Typing: The generated C++ classes inherit from the generic library classes. The generated classes offer a strong typing interface while their generic superclasses offer a weak typing interface to the same objects. The examples in the following text will show how both are intended to be used. A very important feature is that both interfaces can be used for the same objects in a mixed fashion. Therefore, it is possible to use generic components together with strong typing components in the same application.

Meta Information: Every ASN.1 C++ class has access to run time type information (meta information) to support a dynamic style of usage in the application. Generic applications, e.g. a graphic application program, rely on this meta information.

Local Types: Any auxiliary type definition, e.g. the values of an enumeration type or the selector type for alternatives of a choice is defined within the scope of the class which uses it in order to avoid name clashes in the global scope.

Compatibility: All C++ classes for primitive types are made compatible with the corresponding C++ basic type, e.g. ASN.1 integers are compatible with C++ integers.

Qualified Identifiers: The overall convention for generated names is: "⟨ASN.1 Module⟩_⟨ASN.1 type name⟩_⟨ASN.1 component name⟩", where the module name is a nickname in upper case letters, and the type name is the same as in the ASN.1 source text. The component name is only generated for anonymous component types.

Meta Information

The purpose of Meta Information is to provide the type information derived from the various GDMO & ASN.1 specifications which is needed by the OOI at run time. Such information may be used directly or indirectly by applications that

1. use the generic interfaces of the OOI (as opposed to the type-safe interface);

2. offer a generic GUI requiring conversion to/from string format; and

3. display GDMO meta information to a user.

Typically the meta information is not used directly by applications that only use the type safe interface. Internally, the OOI-Run Time System (RTS) makes use of the Meta Information.

The GDMO standard defines templates for the definition of management information. The OOI RTS provides meta information about management information specified using the following templates:

MANAGED OBJECT CLASS which specifies the names of the mandatory and optional packages of a managed object;

PARAMETER which specifies the syntax and behavior of parameters that may be associated with particular attributes, operations and notifications;

ATTRIBUTE which defines admissible operations for the attribute and refers to an ASN.1 type definition;

ATTRIBUTE GROUP which specifies a cluster of attribute that can be accessed or operated upon under one name;

ACTION which refers to an ASN.1 type for outgoing or incoming information; and

NOTIFICATION which refers to an ASN.1 type for the information that is passed with an event notification of the defined type.

A C++ class is defined for each of those templates. An instance of this class is instantiated for each GDMO template defined in the GDMO document parsed by OIC. A single instance of the OOIMetaInfoRepository class serves as anchor for the meta information. Additionally instances of C++ classes are generated for each document, each ASN.1 module, and each ASN.1 Type defined in a GDMO/ASN.1 document.

The OOI Meta Information can be seen as a data structure which holds most of the information contained in GDMO/ASN.1 documents. This information is stored in a set of objects that provide methods to retrieve specific meta information and to 'navigate' through the meta information data structure. For example, the OOIMetaRepository class provides methods to access OOI Document Meta Info and ASN.1 Module Meta Info. Documents Meta Information is stored in a list of object instances. Each instance holds or refers to most of the information contained in one GDMO document. These

instances provide methods to access to Managed Object Class Meta Info, Parameter Meta Info, Attribute Meta Info, Attribute Group Meta Info, Action Meta Info, Notification Meta Info, ASN.1 Meta Info, which are defined in the document.

OOI Error Handling

Three kinds of errors can be encountered when using the OOI:

1. Application Related Errors,

2. OOI Internal Errors,

3. Communication Errors.

Application related errors occur through incorrect coding of the application. Because the OOI supports strong typing, most coding errors will be detected at compil time, but some errors can only be detected at run-time, e.g. trying to set the hour attribute to the unsupported value of 24 or trying to access a bit outside of a string.

OOI internal errors can be caused by system problems (e.g. resource contingency, i.e. out of memory), XMP library or system errors.

Communication Errors are ACSE or CMIS service errors. Those errors are expected as they are an inherent part of the protocol definition.

The OOI uses four mechanisms to signal errors:
Boolean return value;
NULL pointers return value;
Error Objects returned as function return or through function reference arguments; and
Error Objects thrown by exceptions.

Errors Objects is the preferred error handling method of the OOI. Alternatively, booleans or NULL pointers are returned by some functions to provide additional comfort to the application writer.

Boolean return values are used by functions that are typically used in an evaluation.

NULL pointer return values are returned by Meta Info access methods if the meta information is not available.

Error objects are returned through function reference arguments of the called methods.

Exceptions must be used in C++ to handle failing constructors, because constructors do not support result parameter (hence can't return an error code). Furthermore, exceptions provide a convenient mechanism to indicate internal errors. The OOI throws Error Objects of the same classes as those used in reference function arguments.

Error Objects

Error objects are returned by C++ functions through reference arguments or thrown via C++ exceptions. The following figure describes the inheritance hierarchy of OOIErrors. All error classes are derived from the Exeption class of the IBM Application Support Classes.

Communication errors are expected as they are inherent to the ACSE and CMIS protocol operation. Error objects describing communication errors are usually 'returned' and not 'thrown'. Communication errors can originate from the stack interface (currently XMP) or from the CMIS protocol itself.

Methods of OOI Error objects

OOIError objects usually are allocated by the OOI and passed out to the application. It is the duty of the application to delete these objects. Constructors are disabled explicitly, but the application may use the copy() method to create a copy of an existing error object.

The following methods for inspecting the kind of error are provided:
isCommunicationsError() checks if the error is of the category ..
isCMISError() checks if the error is of the category ..
isLocalError() checks if the error is of the category ..
isApplicationError() checks if the error is of the category ..
isInternalError() checks if the error is of the category ..

The following general Support Methods are provided:
the name() method return the name of the objects class
the typeCodeIndex() method returns the index of the sub class
(may be used for inspecting / classifying an error, usually followed by ::narrow)
(for a list of typecodes, which is an enumeration type, look into the header file OOIError.H)
<< the print operator puts a formatted print of the status of the object into an ostream object
the asString() returns a string containing the formatted status of the object
the copy() method returns a pointer to a copy of the parameter object
the metaInfo() method returns a pointer to the meta information of the object
the narrow() method performs a type safe conversion into an instance of a subclass

Implementation of the OOI

The OOI can be implemented either as a program written in the Object Oriented programming language C++ or installed as a hardware tool.

Claims

1. An object-oriented programming interface for developing and running network management applications on a network communication infrastructure, wherein the management applications have access

to and can manipulate managed objects which are available at remote managed agents via the communication infrastructure and can exchange management information between the management applications and the agents; and
wherein the managed objects are defined by object classes (MOC) and specified in an object-oriented syntax notation defining data types;
the programming interface comprising:
object interface composing (OIC) means for generating code which provides proxy managed object (PMO) classes as local representatives for the managed object classes (MOC), object classes for types defined in a syntax notation, notification classes for incoming notifications, and methods to manipulate specified managed objects;
run time system (RTS) means for providing proxy agent object (PAO) classes as representatives for remote agents which provide common management information services (CMIS), to allow access to the CMIS;
wherein the code generated by the OIC is linked to the RTS for execution.

2. Interface according to claim 1, wherein the OIC means generates C++ code, wherein the application programs are written in C++ and wherein the PMO and the ASN classes are object classes in C++.

3. Interface according to claim 1 and/or 2, wherein the OIC means generates automatically PMO C++ class definitions and implementation files based on managed objects written in the ISO standard GDMO and ASN.1.

4. Interface according to one or more of claims 1 to 3, wherein the OIC comprises generating means for the generation of code for managed objects based on the specifications of management information in GDMO and in ASN.1.

5. Interface according to one or more of the preceding claims, wherein managed objects are full embedded into C++ to allow strong type checking at compile time.

6. Interface according to one or more of the preceding claims, wherein incoming notifications are stored in an event queue object.

7. Interface according to one or more of the preceding claims, wherein the PAOs provide a service interface to the operations of CMIS.

8. Interface according to one or more of the preceding claims, wherein providing direct addressing proxy agents (DAPA) which can connect to one specific agent at a time.

9. Interface according to one or more of the preceding claims, wherein further providing non direct addressing proxy agents (NDAPA) for which for each management request the agent must be addressed explicitly by supplying an addressing parameter as part of the request by means of a context object or implicitly through a directory service.

10. Interface according to one or more of the preceding claims, wherein signalling errors by error objects returned as a function return or through function reference arguments and/or error objects thrown by exceptions.

11. Interface according to claim 10, wherein defining error classes which comprise an inheritance hierarchy.

12. Interface according to one or more of claims 9 to 11, wherein the OIC means generates a C++ class for every GDMO managed object class, a C++ class for each ASN.1 type, a C++ class for every GDMO notification, meta information data structures for GDMO and ASN.1, and a set of utility files.

13. Interface according to one or more of the preceding claims, wherein the proxy agents provide a set of the additional methods simple-create, simple-get, simple-action and simple-delete with less and simplified parameter to perform operations on only one attribute of one managed object or on several attributes of a single managed object.

14. Interface according to one or more of the preceding claims, wherein the inheritance relation between generated PMO classes in C++ reflects the inheritance relation of the MOCs defined in GDMO/ASN.1 documents.

15. Interface according to claim 14, wherein the definition of managed object classes is derived from more than one superior class (multiple inheritance).

16. Interface according to one or more of the preceding claims, wherein the RTS receives notifications and stores them in the event queue means of the responsible PAO.

17. A method for mapping of GDMO templates into C++ classes by means of the object-oriented programming interface according to one or more of the preceding claims,
wherein the inheritance hierarchy between managed object classes (MOC) is preserved in C++;
wherein package and attribute templates are merged into MOCs;
wherein access methods to the values of the attributes are generated for the MOCs; and

wherein for every GDMO action a method of the MOC is provided.

18. Method according to claim 17, wherein the OIC means minimizes the number of generated classes and the number of objects to be handled by an application at run time.

19. Method according to claim 17 and/or 18, wherein for every notification template a C++ class is generated.

20. Method according to one or more of claims 17 to 19, wherein for every GDMO MOC meta object instances are generated.

21. Method according to one or more of the preceding claims, wherein no classes are generated for GDMO packages and GDMO attributes.

22. Method according to one or more of the preceding claims, wherein the objects are accessed by a strong typed interface to allow the compiler to detect type errors and/or by a weak typed interface to manipulate objects whose type is not known.

23. A method for mapping from ASN.1 defined types into C++ language by means of the object-oriented programming interface according to one or more of the preceding claims,
wherein generating for every primitive ASN.1 type and every ASN.1 constructor a corresponding class in the ASN.1 C++ library;
wherein each application defined ASN.1 type is mapped to at least one C++ class; and
wherein instances of these classes form a tree structure with instances of ASN.1 constructors as intermediate nodes and instances of primitive ASN.1 types as leaves and wherein the root object of this tree is an application defined class which inherits from the outermost ASN.1 constructor class or simple type.

24. Method according to claim 23, wherein the generated C++ classes and those in the library are derived from a single common class.

25. A network platform for implementing the object-oriented programming interface according to one or more of the preceding claims, wherein the management applications interact with the object-oriented int rface (OOI) through the proxy managed objects (PMO), directly through the proxy agent objects or through the notification event queue.

26. Platform according to claim 25, wherein the OOI uses XMP to access the communication infrastructure which allows it to communicate with an agent that implements the managed objects.

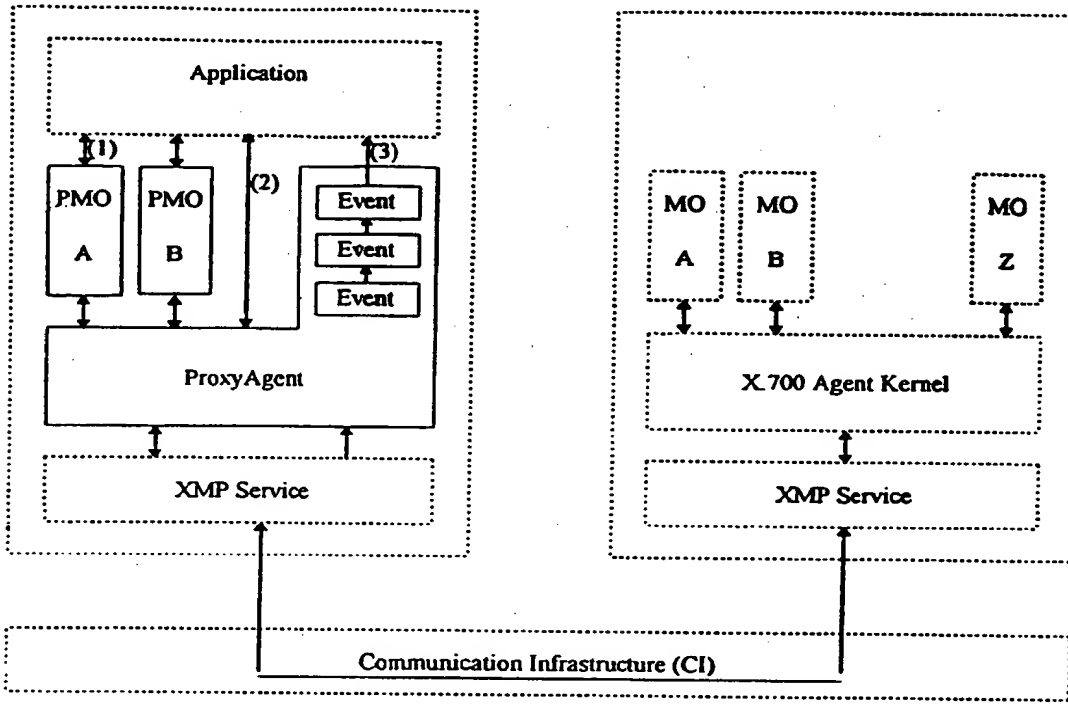

Fig. 1

GDMO Templates	Semantics
Managed Object Class	The basic template for the definition of entities of management information. Mainly references GDMO Package Templates.
Package	Three possibly empty sequences of names of • GDMO Attribute Templates with annotation for accessibility, initial value, and value ranges, • GDMO Action Templates and • GDMO Notification Templates.
Attribute	A reference to another attribute template (and optional modifiers), or a reference to an ASN.1 defined type with an indication on the required operations.
Attribute Group	Clusters of GDMO Attributes to allow to reference several Attributes at once under one name.
Action	Defines a method of the managed object class and optionally references to ASN.1 defined types for outgoing and incoming information.
Notification	References an ASN.1 type for the information which will be passed with the event notification of the defined type. In addition, parts of this ASN.1 structure can be named by Attribute template labels.
Parameter	This template permits the specification and the registration of a parameter syntax and behavior that may be associated with particular attributes, actions and notifications.
Name Binding	Defines the allowed naming and containment relationship between managed objects. This template is not of direct interest for generating source code f r management applicati ns.

Fig. 2

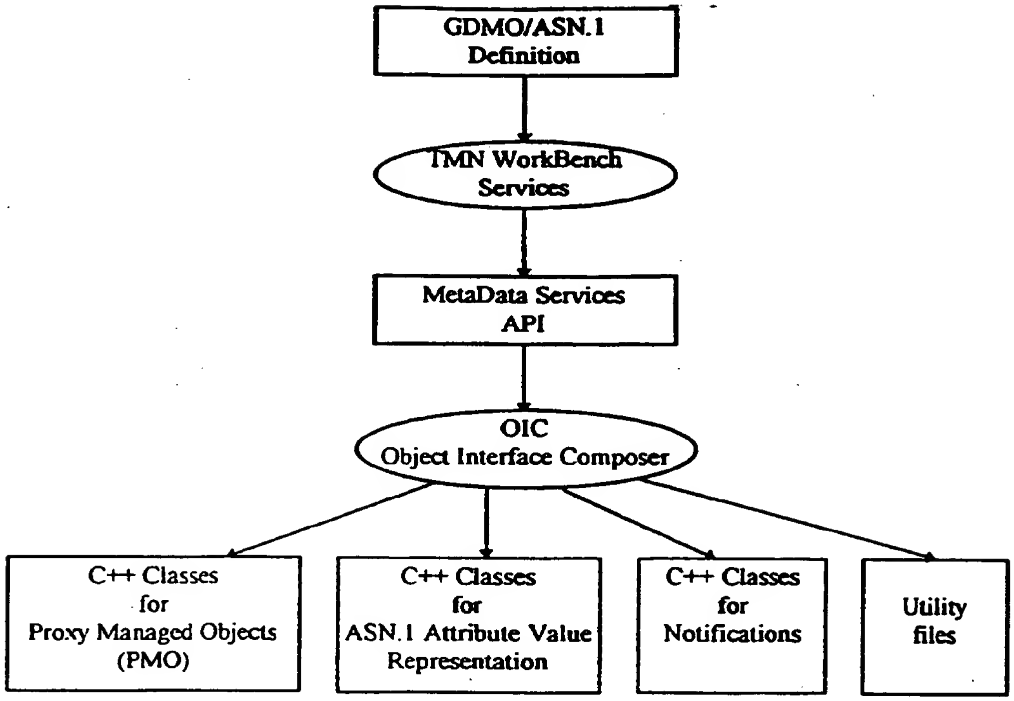
GDMO/ASN.1
Definition
TMN WorkBench
Services
MetaData Services
API
OIC
Object Interface Composer
C++ Classes
for
Proxy Managed Objects
(PMO)
C++ Classes
for
ASN.1 Attribute Value
Representation
C++ Classes
for
Notifications
Utility
files

Fig. 3

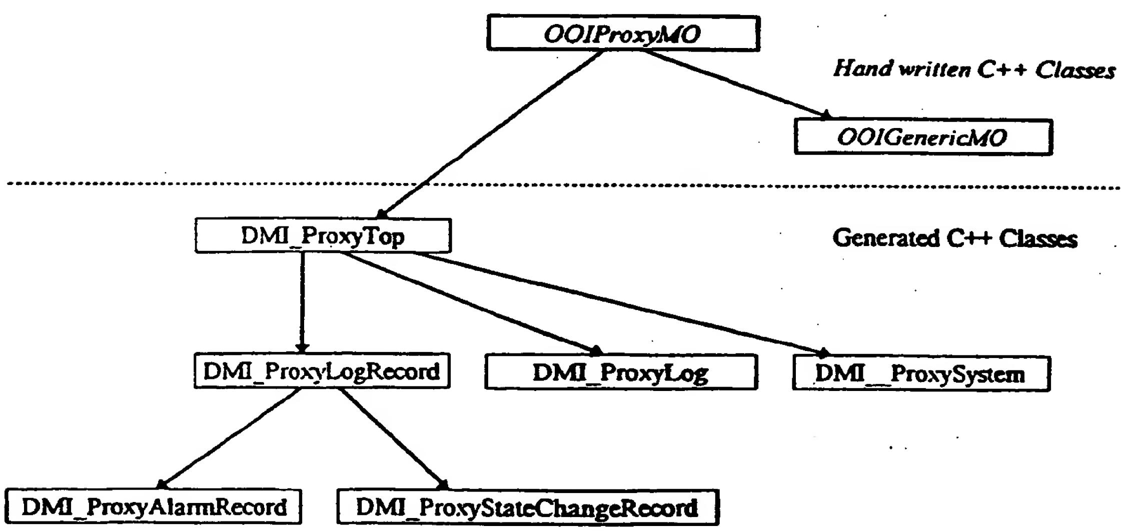
OOIProxyMO
Hand written C++ Classes
OOIGenericMO
DMI_ProxyTop
Generated C++ Classes
DMI_ProxyLogRecord
DMI_ProxyLog
DMI__ProxySystem
DMI_ProxyAlarmRecord
DMI_ProxyStateChangeRecord

Fig. 4

European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 95 10 2234

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
X	OBJECT-ORIENTED SIMULATION CONFERENCE (OOS'93). PROCEEDINGS OF THE 1993 SCS WESTERN SIMULATION MULTICONFERENCE, LA JOLLA, CA, USA, 17-20 JAN. 1993, 1993, SAN DIEGO, CA, USA, SCS, USA, MODIRI N ET AL 'Generic network management system templates' * page 183, left column, line 27 - right column, line 8 * * page 184, left column, line 7 - line 12 * * page 185, left column, line 16 - page 187, left column, line 18 * ---	1-26	G06F9/44
A	INTEGRATED NETWORK MANAGEMENT. IFIP TC6/WG6.6. THIRD INTERNATIONAL SYMPOSIUM, SAN FRANCISCO, CA, USA, 18-23 APRIL 1993, vol. C-12, ISSN 0926-549X, IFIP TRANSACTIONS C (COMMUNICATION SYSTEMS), 1993, NETHERLANDS, pages 593-604, DOSSOGNE F ET AL 'A software architecture for management information model definition, implementation and validation' * page 598, line 1 - line 37 * ---	1-26	TECHNICAL FIELDS SEARCHED (Int.Cl.6) G06F
A	UPPER LAYER PROTOCOLS, ARCHITECTURES AND APPLICATIONS. IFIP TC6/WG6.5 INTERNATIONAL CONFERENCE, VANCOUVER, BC, CANADA, 27-29 MAY 1992, vol. C-7, ISSN 0926-549X, IFIP TRANSACTIONS C (COMMUNICATION SYSTEMS), 1992, NETHERLANDS, pages 195-208, NEWNAN O 'On managing to be managed (OSI systems)' * the whole document * -----	1-26	

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	4 August 1995	Brandt, J

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)

THIS PAGE BLANK (USPTO)